

Visual Basic Language Concepts

Modeling a Real-World Object: Creating Your First Class

[See Also](#)☐ Collapse All Language Filter: Visual Basic

In this lesson, you will learn how to create a class using a **Class Library** project.

In the previous lesson, you learned that classes can be used as blueprints for objects that model real-world things. One of the best reasons for using classes is that once you have created a class for a certain type of object, you can reuse that class in any project.


For example, many programs that you write might involve people—an address-book program for keeping track of your friends, a contact-manager program for your business contacts, or a program for tracking employees. Although the programs may be considerably different, the attributes that apply to a person would be the same. Every person has a name, an age, an address, and a phone number.

In this and the next few lessons, you will create a class that represents a person; you can save this class and use it in other programs that you write in the future.

Classes can be created in three ways—as a part of the code in a form module in a **Windows Application** project, as a separate class module added to a **Windows Application** project, or as a stand-alone **Class Library** project.

☐ Creating Classes


You may have noticed in some of the earlier lessons that when you double-click a form and open the Code Editor, you see something like the following:

 Copy Code

```
Public Class Form1
    Private Sub Form1_Load...

    End Sub
End Class
```

That's right—the form is actually a class, marked by **Class** and **End Class** statements, and any code that you enter between those two statements is part of the class. Although by default a form module contains only a single class, you could create additional modules by adding code below the **End Class** statement as follows:

 Copy Code

```
Public Class Form1
    ' Form1 code here
End Class
Public Class MyFirstClass
    ' Your class code here
End Class
```

The drawback to creating classes this way is that they are available only within the project in which you created them. If you want to share a class with other projects, you'll want to put it in a class module.

☐ Class Modules

A class module is a separate code file that contains one or more classes. Because it is a separate file, it can be reused in other projects. Class modules can be created in two ways—as a module added to a **Windows Application** project, or as a stand-alone **Class Library** project.

You can add a new class module to an existing project by selecting **Class** in the **Add New Item** dialog box, available from the **Project** menu. For the purpose of this set of lessons, you will be creating a stand-alone **Class Library** project.

Try It!

To create a Class Library project

1. On the **File** menu, choose **New Project**.
2. On the **Templates** pane, in the **New Project** dialog box, click **Class Library**.
3. In the **Name** box, type `Persons` and then click **OK**.
A new **Class Library** project opens, and the Code Editor displays the `class` module `Class1.vb`.
4. In **Solution Explorer**, right-click `Class1.vb` and select **Rename**, and then change the name to `Persons.vb`.
Notice that the name in the Code Editor also changed to `Persons.vb`.
5. On the **File** menu, choose **Save All**.
6. In the **Save Project** dialog box, click **Save**.

Tip

Rather than saving the project in the default location, you may want to create a directory where you can store all of your classes for reuse. You can enter that folder in the **Location** field of the **Save Project** dialog box before you save.

For now, keep the project open—you will add to it in the next lesson.

Next Steps

In this lesson, you learned how to create a class module. An empty class is of little use though—in the next lesson, you will learn how to add properties to your class.

Next Lesson: [Adding Properties to Your Class](#)

See Also

Tasks

[How to: Add New Project Items](#)

Concepts

[What is a Class?](#)

Other Resources

[Programming With Objects: Using Classes](#)

To make a suggestion or report a bug about Help or another feature of this product, go to the [feedback site](#).

Visual Basic Language Concepts

Adding Properties to Your Class

[See Also](#)☐ Collapse All ☒ Language Filter: Visual Basic

In this lesson, you will learn how to add properties to the class that you created in the previous lesson.

In an earlier lesson, [Closer Look: Understanding Properties, Methods, and Events](#), you learned that all objects have attributes, and that properties represent attributes. In the previous lesson, you created a `Persons` class that represents a person; people have attributes such as name and age, so the `Persons` class needs properties to represent those attributes.


Properties can be added to a class in one of two ways: as a *field*, or as a *property procedure*. You can also determine how a property works by using the **Public**, **ReadOnly**, or **WriteOnly** modifiers for the property.

☒ Fields and Property Procedures

Fields are really just public variables within a class that can be set or read from outside of the class. They are useful for properties that don't need to be validated—for example, a **Boolean** (**True** or **False**) value. In the case of the `Persons` class, you might have a **Boolean** property named `Alive` that specifies whether a person is dead or alive. Since there are only two possible values, a field works well for this property.

To add a field to a class, the code would look like the following.

Visual Basic


 Copy Code

```
Public Alive As Boolean
```

Most properties, however, are more complex than that—in most cases you will want to use a property procedure to add a property to a class. Property procedures have three parts: a declaration of a private variable to store the property value; a **Get** procedure that exposes the value; and a **Set** procedure that, as it sounds, sets the value.

For example, a property procedure for a `Name` property for the `Persons` class would look like the following.

Visual Basic

 Copy Code

```
Private nameValue As String
Public Property Name() As String
    Get
        Name = nameValue
    End Get
    Set(ByVal value As String)
        nameValue = value
    End Set
End Property
```

The first line of code declares a private **String** variable, `nameValue`, which will store the value of the property. The property procedure itself begins with `Public Property` and ends with `End Property`.

The **Get** procedure contains the code that will be executed when you want to read its value—for example, if you read the `Persons.Name` property, the code would return the value stored in the `nameValue` variable.

The **Set** procedure contains code used to assign a new value to the `nameValue` variable using a value passed to it as a `value` argument. For example, if you wrote the code `Persons.Name = "John"`, the **String** value `John` would be passed as the `value` argument; the code in the **Set** procedure would then assign it to the `NameValue` variable for storage.

You might ask why you would go to all that trouble rather than use a field to represent the `Name` property. In the real world, there are certain rules for names—for example, names do not usually

contain numbers. You could add code to the **Set** procedure to check the `value` argument and return an error if it contains numbers.


In the following procedure, you will add a field and three properties to the `Persons` class.

Try It!

To add properties to your class

1. Open the `Persons` project that you created in the previous lesson. If you did not save it, you will first need to go back to the previous lesson, [Modeling a Real-World Object: Creating Your First Class](#), and complete the procedures in that lesson.
2. In **Solution Explorer**, select **Persons.vb**, and then on the **View** menu, choose **Code**.
3. Add the following declaration code below the `Public Class Persons` line.


Visual Basic

 Copy Code

```
Private firstNameValue As String
Private middleNameValue As String
Private lastNameValue As String
Public Alive As Boolean
```

4. Add the following property procedures below the declaration code.

Visual Basic

 Copy Code

```
Public Property FirstName() As String
    Get
        FirstName = firstNameValue
    End Get
    Set(ByVal value As String)
        firstNameValue = value
    End Set
End Property

Public Property MiddleName() As String
    Get
        MiddleName = middleNameValue
    End Get
    Set(ByVal value As String)
        middleNameValue = value
    End Set
End Property

Public Property LastName() As String
    Get
        LastName = lastNameValue
    End Get
    Set(ByVal value As String)
        lastNameValue = value
    End Set
End Property
```

5. On the **File** menu, choose **Save All** to save your work.


Read-only and Write-only Properties

Sometimes a property is meant to be set once and never changed during the execution of your program. For example, a property representing an employee number should never change, so it could be read by another program but you would not allow that program to change its value.

The **ReadOnly** keyword is used to specify that a property value can be read but not modified. If you try to assign a value to a **ReadOnly** property, an error occurs in the Code Editor.

To create a read-only property, you would create a property procedure with a **Get** procedure but no **Set** procedure, as follows.

Visual Basic

 Copy Code


```


Private IDValue As Integer
ReadOnly Property ID() As Integer
    Get
        ID = IDValue
    End Get
End Property

```

Likewise, the **WriteOnly** keyword allows a property value to be set but not read—for example, you would not allow a password property to be read by other programs. You might use that value to do things within your class, but you would want to keep it private.

To create a write-only property, you would create a property with a **Set** procedure but no **Get** procedure, as follows.

Visual Basic

 Copy Code

```

Private passwordValue As String
WriteOnly Property Password() As String
    Set(ByVal value As String)
        passwordValue = value
    End Set
End Property

```

ReadOnly and **WriteOnly** property procedures are also useful when you want to take one property value and convert it to a different value. For example, consider a person's age. Unlike a name, age changes over time—if you assigned your age to a class and read it back a year later, it would be wrong.


In the `Persons` class, you could prevent this by adding two properties—a **WriteOnly** `BirthYear` property that represents the year of your birth, which never changes, and a **ReadOnly** `Age` property that returns a value by calculating the difference between the current year and your birth year.

Try It!

To add **ReadOnly** and **WriteOnly** properties to your class

1. Add the following declaration code below the other declarations at the top of the class module.


Visual Basic

 Copy Code

```
Private birthYearValue As Integer
```

2. Add the following property procedures below the declaration code.

Visual Basic

 Copy Code

```

WriteOnly Property BirthYear() As Integer
    Set(ByVal value As Integer)
        birthYearValue = value
    End Set
End Property

ReadOnly Property Age() As String
    Get
        Age = My.Computer.Clock.LocalTime.Year - birthYearValue
    End Get
End Property

```

3. On the **File** menu, choose **Save All** to save your work.

Next Steps

In this lesson, you learned about properties and the different ways to add them to your class. In the next lesson, you will learn how to add methods to your class so that it can perform actions.

Next Lesson: [Adding Methods to Your Class](#)

See Also

Tasks

[Modeling a Real-World Object: Creating Your First Class](#)

Concepts

[Property Procedures vs. Fields](#)



Other Resources

[Programming With Objects: Using Classes](#)

To make a suggestion or report a bug about Help or another feature of this product, go to the [feedback site](#).

Visual Basic Language Concepts

Adding Methods to Your Class

[See Also](#) Collapse All  Language Filter: Visual Basic


In this lesson, you will learn how to add methods to a class so that it can perform actions.

In an earlier lesson, [Closer Look: Understanding Properties, Methods, and Events](#), you learned that most objects have actions that they can perform; these actions are known as methods. The `Persons` class that you created in the [Modeling a Real-World Object: Creating Your First Class](#) lesson represents a person. There are many actions that people can perform, and for the `Persons` class, those actions can be expressed as class methods.

Class Methods

The methods of a class are simply **Sub** or **Function** procedures declared within the class. For example, an `Account` class might have a **Sub** procedure named `Recalculate` that would update the balance, or a `CurrentBalance` **Function** procedure to return the latest balance. The code to declare those methods might look like the following:

Visual Basic


 Copy Code

```
Public Sub Recalculate()  
    ' add code to recalculate the account.  
End Sub  
Public Function CurrentBalance(ByVal AccountNumber As Integer) As Double  
    ' add code to return a balance.  
End Function
```

While most class methods are public, you might also want to add methods that can only be used by the class itself. For example, the `Persons` class might have its own function for calculating a person's age. By declaring the function as **Private**, it can not be seen or called from outside the class.

The code for a private function might look like the following:

Visual Basic

 Copy Code

```
Private Function CalcAge(ByVal year As Integer) As Integer  
    CalcAge = My.Computer.Clock.LocalTime.Year - year  
End Function
```

You could later change the code that calculates the value of `CalcAge`, and the method would still work fine without changing any code that uses the method. Hiding the code that performs the method is known as *encapsulation*.


In the `Persons` class, you will create a public method that returns a full name, and a private function to calculate the age.

Try It!

To add a method to your class

1. Open the `Persons` project that you created in the previous lesson. If you did not save it, you will first need to go back to the previous lesson, [Adding Properties to Your Class](#), and complete the procedures.
2. In **Solution Explorer**, select **Persons.vb**, and then on the **View** menu choose **Code**.
3. Add the following code below the property procedures.


Visual Basic

 Copy Code

```
Public Function FullName() As String  
    If middleNameValue <> "" Then  
        FullName = firstNameValue & " " & middleNameValue & " " &  
            & lastNameValue
```




```
Else
    FullName = firstNameValue & " " & lastNameValue
End If
End Function
```

Visual Basic Copy Code

```
Private Function CalcAge(ByVal year As Integer) As Integer
    CalcAge = My.Computer.Clock.LocalTime.Year - year
End Function
```

4. Modify the code in the Age property procedure to use the private function.

Visual Basic Copy Code

```
ReadOnly Property Age() As String
    Get
        ' Age = My.Computer.Clock.LocalTime.Year - birthDateValue
        Age = CalcAge(birthYearValue)
    End Get
End Property
```

5. On the **File** menu, choose **Save All** to save your work.

▣ Next Steps

In this lesson, you learned how to add both public and private methods to a class. You can learn more about methods in [Closer Look: Creating Multiple Versions of the Same Method with Overloading](#), or you can go on to the next lesson and learn how to use and test the class you created.

Next Lesson: [Testing Your Class](#)

▣ See Also**Tasks**

[Adding Properties to Your Class](#)

Other Resources

[Programming With Objects: Using Classes](#)

To make a suggestion or report a bug about Help or another feature of this product, go to the [feedback site](#).

Visual Basic Language Concepts

Closer Look: Creating Multiple Versions of the Same Method with Overloading

[See Also](#)☐ Collapse All Language Filter: Visual Basic

In this lesson, you will learn how to add multiple versions of a method to your class.

In the previous lesson, you learned how to add methods to the `Persons` class. Sometimes there are cases where a single method will not do—for example, you might need to pass different data types to the method in different situations, or you might want to return different formats as a result.


You can create multiple versions of a method using a technique called *overloading*. When a class has more than one method with the same name but with a different set of arguments, the method is overloaded.

☐ Overloading

To create an overloaded method, add two or more **Sub** or **Function** procedures to your class, each with the same name. In the procedure declarations, the set of arguments for each procedure must be different or an error will occur.

The following shows a method with two overloads, one which takes a **String** and the other which takes an **Integer** as an argument.

Visual Basic

 Copy Code

```
Public Sub TestFunction(ByVal input As String)
    MsgBox(input)
End Sub
Public Sub TestFunction(ByVal input As Integer)
    MsgBox(CStr(input))
End Sub
```

If you were to call this method from your code and pass it a string, the first overload would be executed and a message box would display the string; if you passed it a number, the second overload would be executed, and the number would be converted to a string and then displayed in the message box.

You can create as many overloads as you need, and each overload can contain a different number of arguments.


In the `Persons` class, you will add a method with two overloads to return a person's middle initial: one with just the initial, the other with the initial followed by a period.

☐ Try It!

To create an overloaded method

1. Open the `Persons` project that you created in the previous lesson. If you did not save it, go back to the previous lesson, [Adding Methods to Your Class](#), and complete the procedures.
2. In **Solution Explorer**, select **Persons.vb**, and then on the **View** menu, choose **Code**.
3. Add the following code below the existing methods.

Visual Basic

 Copy Code

```
Public Function MiddleInitial() As String
    MiddleInitial = Left$(middleNameValue, 1)
End Function

Public Function MiddleInitial(ByVal period As Boolean) As String
    MiddleInitial = Left$(middleNameValue, 1) & "."
End Function
```

4. On the **File** menu, choose **Save All** to save your work.

☐ Next Steps

In this lesson, you learned how to create an overloaded method. In the next lesson, you will learn how to use the class that you created in a test project.

Next Lesson: [Testing Your Class](#)

See Also

Tasks

[Adding Methods to Your Class](#)

Concepts

[Considerations in Overloading Procedures](#)

Other Resources

[Programming With Objects: Using Classes](#)

[Visual Basic Guided Tour](#)

To make a suggestion or report a bug about Help or another feature of this product, go to the [feedback site](#).

Visual Basic Language Concepts

Testing Your Class

[See Also](#)☐ Collapse All Language Filter: Visual Basic


In this lesson, you will learn how to create an instance of a class in order to test the class.

In the past few lessons, you have created a `Persons` class and given it properties and methods. So far, all you've done is add code—now it is time to use the `Persons` class and make sure that it works as expected.

☐ Creating an Instance of a Class

Although you may not have realized it, you have been using classes in many of the previous lessons. Forms and controls are actually classes; when you drag a **Button** control onto a form, you are actually creating an instance of the `Button` class.

Any class can also be instantiated in your code using a declaration with the **New** keyword. For example, to create a new instance of the **Button** class, you would add the following code.

Visual Basic Copy Code


```
Dim aButton As New Button
```

To use and test the `Persons` class, you must first create a test project and add a reference to the class module.

☐ Try It!

To create a test project for your class

1. Open the `Persons` project that you created in the previous lesson. If you did not save it, you will first need to go back to the previous lesson, [Adding Methods to Your Class](#), and complete the procedures.
2. On the **File** menu, point to **Add**, and choose **New Project**.
3. On the **Templates** pane in the **New Project** dialog box, click **Windows Application**.
4. In the **Name** box, type `PersonsTest` and then click **OK**.
5. In **Solution Explorer**, select the `PersonsTest` project, and then on the **Project** menu, choose **Set as StartUp Project**.
6. In the **Solution Explorer**, select the `PersonsTest` project, and then on the **Project** menu, choose **Add Reference**.
The Add Reference dialog box opens.
7. Click the **Projects** tab, and then select **Persons** and click **OK**.
8. Double-click the form to open the Code Editor, and then enter the following declaration just below the line `Public Class Form1`.

Visual Basic Copy Code

```
Dim person1 As New Persons.Persons
```

This declares a new instance of the `Persons` class. You might wonder why you needed to type `Persons` twice—the first instance is the `Persons.vb` class module; the second instance is the `Persons` class within that module.

9. On the **File** menu, choose **Save All**.

☐ Testing Your Class


The next step is to add a user interface and code that uses the `Persons` class. You will add text boxes into which the user will enter values for each of the properties (except the read-only `Age` property), a check box for the `Alive` field, and buttons to test each of the public methods.

☐ Try It!

To test your class

1. In **Solution Explorer**, select **Form1**, and then on the **View** menu, choose **Designer**.
2. From the **Toolbox**, drag four **TextBox** controls, a **CheckBox** control, and two **Button** controls onto the form.
3. **Select the first Button control, and then in the Properties window set its Text property to Update.**
4. **Select the second Button control, and then in the Properties window set its Text property to Full Name.**
5. Double-click the first button (**Update**) to open the Code Editor, and then in the **Button1_Click** event handler, add the following code.

Visual Basic


 Copy Code

```
With person1
    .FirstName = Textbox1.Text
    .MiddleName = Textbox2.Text
    .LastName = Textbox3.Text
    .BirthYear = Textbox4.Text
    .Alive = CheckBox1.Checked
End With
```

Notice that as you type, a list containing all of the members of the **Persons** class is displayed. Since it was added as a reference, IntelliSense displays information about your class just as it would for any other class.

6. In the **Button2_Click** event handler, add the following code.

Visual Basic

 Copy Code

```
' Test the FullName method.
MsgBox(person1.FullName)

' test the Age property and CalcAge method.
MsgBox(CStr(person1.Age) & " years old")

' Test the Alive property.
If person1.Alive = True Then
    MsgBox(person1.FirstName & " is alive")
Else
    MsgBox(person1.FirstName & " is no longer with us")
End If
```

7. Press F5 to run the project and display the form.
 - a. In the first text box, enter your first name.
 - b. In the second text box, enter your middle name.
 - c. In the third text box, enter your last name.
 - d. In the fourth text box, enter the four-digit year you were born (i.e. 1983).
 - e. If you are still alive, select that check box.
8. Click the **Update** button to set the properties of the class, and then click the **Full Name** button.
Three message boxes are displayed, showing your full name, your age, and your status.
9. On the **File** menu, choose **Save All**.

Testing the Overloaded Methods

If you completed the optional lesson [Closer Look: Creating Multiple Versions of the Same Method with Overloading](#), you will also want to test the overloaded methods that you added to the **Persons** class. If you did not complete the lesson, you can go back and do so now, or you can skip the following procedure.

Try It!


To test the overloaded methods

1. In **Solution Explorer**, select **Form1**, and then on the **View** menu, choose **Designer**.
2. From the **Toolbox**, drag two more **Button** controls onto the form.
3. Select the third **Button** control, and then in the **Properties** window set its **Text** property to

With.

4. Select the fourth **Button** control, and then in the **Properties** window set its **Text** property to **Without**.
5. Double-click the first button (**With**) to open the Code Editor, and then enter the following code in the **Button3_Click** event handler.

Visual Basic


 Copy Code

```
MsgBox(person1.FirstName & " " & person1.MiddleInitial(True) & _  
        " " & person1.LastName)
```

Notice that as you type, a list containing all of the members of the `Persons` class is displayed. Since it was added as a reference, IntelliSense displays information about your class, just as it would for any other class.

6. In the **Button4_Click** event handler, add the following code.

Visual Basic

 Copy Code

```
MsgBox(person1.FirstName & " " & person1.MiddleInitial & _  
        " " & person1.LastName)
```

7. Press F5 to run the project and display the form.
 - a. In the first text box, type your first name.
 - b. In the second text box, type your middle name.
 - c. In the third text box, type your last name.
 - d. In the fourth text box, type the four-digit year you were born (i.e. 1983).
 - e. If you are still alive, select the check box.
8. Click the **Update** button to set the properties of the class, and then click the **With** button. A message box displays showing your name with a period after the middle initial.
9. Click the **Without** button. A message box displays showing your name without a period after the middle initial.
10. On the **File** menu, choose **Save All**.

Next Steps

In this lesson, you learned how to create a test project and then use it to test the properties and methods of your class. In the next lesson, you will learn how to use inheritance to create a class based on an existing class.

Next Lesson: [Building Your Class on an Existing Class: Using Inheritance](#).

See Also

Tasks

[Adding Methods to Your Class](#)

[Closer Look: Creating Multiple Versions of the Same Method with Overloading](#)

To make a suggestion or report a bug about Help or another feature of this product, go to the [feedback site](#).

QUOTATION-CMR CMR NO. 4001595

PROJECT 4001163

END USER : WINC

FREIGHT:
ALL TAXES EXTRA WHERE APPLICABLE.

QTY		PRODUCT NO./DESCR	BPC	BASE M/ PRICE D	PRICE PER	UNIT COST	EXT. COST	TOTAL
		120v c1d2 c/b						5.00
2	500	8050	LC	1.00 M	1.00 C	1.00	5.00	5.00
		VALUE ADDED LABOUR						
		S E C T I O N	T O T A L	:			5.00	5.00 *
			T O T A L	:			5.00	5.00 **

Visual Basic Language Concepts

Building Your Class on an Existing Class: Using Inheritance[See Also](#)☐ Collapse All Language Filter: Visual Basic

In this lesson, you will learn how to use inheritance to create a class based on an existing class.


Many real-life objects have attributes and behaviors in common—for example, all cars have wheels and engines, and can roll and (hopefully) stop. Some cars, however, have attributes that are not common—for example, a convertible has a removable top, which may be lowered electronically or by hand.

If you created an object to represent a car, you would want to include properties and methods for all of the common attributes and behaviors, but you would not want to add attributes such as a convertible top, since that attribute does not apply to all cars.

Using *inheritance*, you can create a "convertible" class that is derived from the car class. It inherits all of the attributes of the car class, and it can add those attributes and behaviors that are unique to a convertible.

☐ Inheriting From an Existing Class

The **Inherits** statement is used to declare a new class, called a *derived class*, based on an existing class, known as a *base class*. Derived classes inherit all of the properties, methods, events, fields, and constants defined in the base class. The following code shows the declaration for a derived class.

Visual Basic Copy Code


```
Class DerivedClass
    Inherits BaseClass
End Class
```

The new class, `DerivedClass`, can then be instantiated, its properties and methods accessed just like `BaseClass`, and you can add new properties and methods that are specific to the new class. For an example, look at the `Persons` class that you created in the previous lessons.

Suppose you wanted a class that represented baseball players—baseball players have all of the attributes defined in the `Persons` class, but they also have unique attributes such as number and position. Rather than adding those properties to the `Persons` class, you will create a new derived class that inherits from `Persons`, and add the new properties to that class.


☐ Try It!**To create a derived class**

1. Open the `Persons` project that you created in the previous lesson. If you did not save it, go back to, [Testing Your Class](#), and complete the procedures.
2. In **Solution Explorer**, select the **Persons** project node.
3. On the **Project** menu, choose **Add Class**.
4. In the **Add New Item** dialog box, type `Players` in the **Name** box, and then click **Add**. A new class module is added to the project.
5. In the Code Editor, add the following just below the `Public Class Players` line.

Visual Basic Copy Code

```
Inherits Persons
```

6. Add the following code to define two new properties.

Visual Basic Copy Code

```
Private numberValue As Integer
Private positionValue As String
Public Property Number() As Integer
    Get
        Number = numberValue
```



```

        End Get
        Set(ByVal value As Integer)
            numberValue = value
        End Set
    End Property
    Public Property Position() As String
        Get
            Position = positionValue
        End Get
        Set(ByVal value As String)
            positionValue = value
        End Set
    End Property

```

7. On the **File** menu, choose **Save All**.


Testing the Players Class

You have now created a Players class derived from the Persons class. In the following procedure you will create a new program to test the Players class.

To create a test project for your class

1. On the **File** menu, point to **Add**, and then choose **New Project**.
2. In the **Add New Project** dialog box, in the **Templates** pane, select **Windows Application**.
3. In the **Name** box, type `PlayerTest` and then click **OK**.
4. A new Windows Forms project is added to **Solution Explorer**, and a new form displays.
5. In **Solution Explorer**, select the `PlayerTest` project, and then on the **Project** menu, choose **Set as Startup Project**.
6. In **Solution Explorer**, select the `PlayerTest` project, and then on the **Project** menu, choose **Add Reference**.
The **Add Reference** dialog box opens.
7. Click the **Projects** tab and choose **Persons**, and then click **OK**.
8. Double-click the form to open the Code Editor, and then enter the following declaration just below the line `Public Class Form1`.

Visual Basic

 Copy Code

```

Dim player1 As New Persons.Players
Dim player2 As New Persons.Players


```

9. This declares two new instances of the Players class.
10. On the **File** menu, choose **Save All**.

To test the derived class

1. In **Solution Explorer**, select `Form1` in the `PlayerTest` project, and then on the **View** menu, choose **Code**.
2. In the Code Editor, add the following code to the **Form1_Load** event procedure.

Visual Basic

 Copy Code


```

With player1
    .FirstName = "Andrew"
    .LastName = "Cencini"
    .Number = 43
    .Position = "Shortstop"
End With
With player2
    .FirstName = "Robert"
    .LastName = "Lyon"
    .Number = 11
    .Position = "Catcher"
End With

```

3. In **Solution Explorer**, select `Form1` in the `PlayerTest` project, and then on the **View** menu, choose **Designer**.


4. From the **Toolbox**, drag two **Button** controls onto the form.
5. Select the first **Button** control, and then in the **Properties** window, set its **Text** property to `At Bat`.
6. Select the second **Button** control, and then in the **Properties** window, set its **Text** property to `On Deck`.
7. Double-click the first button (`At Bat`) to open the Code Editor, and then enter the following code in the **Button1_Click** event handler.

Visual Basic Copy Code

```
MsgBox(player1.Position & " " & player1.FullName & ", #" & _  
    CStr(player1.Number) & " is now at bat.")
```

Notice that you are using the `FullName` method, which was inherited from the base class `Persons`.

8. In the **Button2_Click** event handler, add the following code.

Visual Basic Copy Code

```
MsgBox(player2.Position & " " & player2.FullName & ", #" & _  
    CStr(player2.Number) & " is on deck.")
```

9. Press F5 to run the program. Click each button to see the results.
10. On the **File** menu, choose **Save All**.

▣ Next Steps

In this lesson, you learned how to inherit from an existing class and how to extend the derived class. You can learn more about inheritance in [Closer Look: Overriding Members](#), or you can go on to the next lesson and learn about collections.

Next Lesson: [Keeping Track Of Things: Using Collections to Manage Multiple Objects](#)

▣ See Also**Concepts**

[Inheritance Basics](#)

To make a suggestion or report a bug about Help or another feature of this product, go to the [feedback site](#).

Visual Basic Language Concepts

Keeping Track Of Things: Using Collections to Manage Multiple Objects

[See Also](#)☐ Collapse All ☒ Language Filter: Visual Basic

In this lesson, you will learn how to use a collection to manage groups of objects.

In an earlier lesson, you learned about using arrays to manage groups of variables. While you could use arrays to manage groups of objects, Visual Basic also has a special type of object called a *collection* that can be used to store and retrieve groups of like objects.


Like an array, each item in a [Collection](#) object has an index that can be used to identify that item. In addition, each item in a **Collection** object has a *key*, a **String** value that can be used to identify the item. The advantage to using a key is that you do not need to remember the index of an item; instead you can refer to it using a meaningful name.

Creating a Collection

Collections are useful when your program uses multiple instances of the same class. For example, look at the **Players** class that you created in a previous lesson. It is likely that you need multiple **Players** objects to represent a baseball team.

The first step in creating a collection is to create an instance of a **Collection** object, as shown in the following declaration.


Visual Basic

 Copy Code

```
Dim baseballTeam As New Collection
```

Once you create the **Collection** object, you can use the **Add** method to add items to it, and the **Remove** method to delete the items. When adding items, first specify the item to add, and then specify the **String** value to be used as a key.

Visual Basic

 Copy Code

```
baseballTeam.Add(playerObject, "Player's Name")
```

When removing an item, use the key to specify the item to remove.

Visual Basic

 Copy Code

```
baseballTeam.Remove("Player's Name")
```


In the following procedure, you will add two new **Players** objects, and then create a team collection and add the **Players** objects to it, using the **Position** property as a key.

Try It!

To create a collection of objects

1. Open the **Persons** project that you created in the previous lesson. If you did not save it, go back to the previous lesson, [Building Your Class on an Existing Class: Using Inheritance](#), and complete the procedures.
2. In **Solution Explorer**, in the **PlayerTest** project, select the **Form1.vb** node, and then on the **View** menu, choose **Code**.
3. In the Code Editor, add the following to the declarations section (below the declaration for **player2**).


Visual Basic

 Copy Code

```
Dim player3 As New Persons.Players  
Dim player4 As New Persons.Players  
Dim team As New Collection
```

4. Add the following code to the **Form1_Load** event procedure.

Visual Basic

 Copy Code


```

With player3
    .FirstName = "Eduardo"
    .LastName = "Saavedra"
    .Number = 52
    .Position = "First Base"
End With

With player4
    .FirstName = "Karl"
    .LastName = "Jablonski"
    .Number = 22
    .Position = "Pitcher"
End With

team.Add(player1, player1.Position)
team.Add(player2, player2.Position)
team.Add(player3, player3.Position)
team.Add(player4, player4.Position)


```

5. In **Solution Explorer**, in the **PlayerTest** project, select the **Form1.vb** node. Then on the **View** menu, choose **Designer**.
6. From the **Toolbox**, drag a **ComboBox** control onto the form.
7. In the **Properties** window, select the **Items** property and click the ... button.
8. In the **String Collection Editor**, enter the following, and then click **OK**.


```

Catcher
First Base
Pitcher
Shortstop

```
9. Double-click the **ComboBox** control to open the Code Editor, and then enter the following code in the `ComboBox1_SelectedIndexChanged` event handler.

 Copy Code

```

Dim SelectedPlayer As Persons.Players
SelectedPlayer = team(ComboBox1.SelectedItem)
MsgBox("Playing " & ComboBox1.SelectedItem & " is " & _
    SelectedPlayer.FullName & "!")

```

10. Press F5 to run the program. Select a position from the drop-down list—the player for that position is displayed in a message box.

Next Steps

In this lesson, you learned how to use a **Collection** object to manage a group of objects. At this point, you can learn more about collections in [Closer Look: Using a For Each...Next Loop in a Collection](#), or you can go on to the next group of lessons and learn about creating your own controls.

Next Lesson: [Visible Objects: Creating Your First User Control](#)

See Also

Tasks

[Closer Look: Using a For Each...Next Loop in a Collection](#)

[Closer Look: Overriding Members](#)

[Building Your Class on an Existing Class: Using Inheritance](#)

Concepts

[Visual Basic Collection Class](#)

Other Resources

[Visible Objects: Creating Your First User Control](#)

To make a suggestion or report a bug about Help or another feature of this product, go to the [feedback site](#).